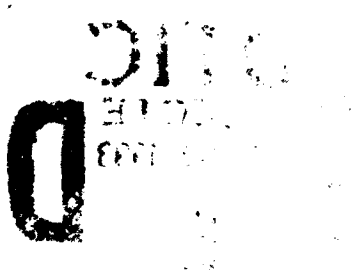
Collection-oriented Match: Scaling Up the Data in Production Systems

Anurag Acharya and Milind Tambe

December 1992
CMU-CS-92-218

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. government.



Keywords: Artificial Intelligence, Database rule systems, Collection-Oriented Match, Set-Oriented rule languages, Scalable match algorithms

Abstract

Match algorithms that are capable of handling large amounts of data, without giving up expressiveness are a key requirement for successful integration of relational database systems and powerful rule-based systems. Algorithms that have been used for database rule systems have usually been unable to support large and complex rule sets, while the algorithms that have been used for rule-based expert systems do not scale well with increasing amounts of data. Furthermore, these algorithms do not provide support for collection (or set) oriented production languages.

This paper proposes a basic shift in the nature of match algorithms: from *tuple-oriented* to *collection-oriented*. A collection-oriented match algorithm matches each condition in a production with a collection of tuples and generates *collection-oriented instantiations*, i.e., instantiations that have collection of tuples corresponding to each condition in the production. This approach shows great promise for efficiently matching expressive productions against large amounts of data. In addition, it provides direct support for collection-oriented production languages. We have found that many existing tuple-oriented match algorithms can be easily transformed to their collection-oriented analogues. This paper presents the transformation of Rete to Collection Rete as an example and compares the two based on a set of benchmarks. Results presented in this paper show that, for large amounts of data, a relatively underoptimized implementation of Collection Rete achieves orders of magnitude improvement in time and space over an optimized version of Rete. The results establish the feasibility of collection-oriented match for integrated database-production systems.

1. Introduction

The integration of relational database systems and production systems (forward-chaining rule systems) promises a number of benefits. Production rules have long been known to be a natural mechanism for enforcing integrity constraints, performing authorization checks and maintaining *views* (derived data) [9]. They have also been used to implement *alerters* that monitor conditions and *triggers* that conditionally initiate actions [7, 28]. A number of prototype relational database systems — STARBURST [34], POSTGRES [28], ARIEL [16] and RPL [8] — use production rules for some or all of these purposes. Commercial relational database systems like INGRES [17] and Sybase [29] also support production systems. Most of these systems use fairly simple match algorithms to determine the set of production instantiations (ARIEL is an exception). This effectively limits the complexity of the rules that can be efficiently supported in the presence of large amounts of data [28]. Typical rules supported in these systems have a small number of simple tests. Supporting powerful rules in a database environment requires match algorithms that can efficiently handle complex tests in the presence of large amounts of data.

On the other hand, expert systems have long supported powerful rules. These systems have traditionally used more powerful match algorithms like Rete and its derivatives [3, 10] and Treat [21]. These algorithms, however, have not been designed for matching large amounts of data and do not scale well [23, 33]. This limits the the amount of data that expert systems can deal with and forces the expert systems that have been coupled with database systems to either use only simple rules or maintain a small separate subset of data by periodically issuing queries [23]. Extending the scope of expert systems to include data-intensive tasks, without giving up expressiveness, requires powerful match algorithms that can efficiently match large amounts of data.

The question then is: can production match algorithms support a large number of powerful match operations and yet scale well with increasing amounts of data ? In this paper, we attempt to answer this question by investigating an approach that avoids the major limitation on the scalability of traditional match algorithms.

The primary reason why traditional match algorithms used in expert systems don't scale well is that they generate a large number of combinations of individual tuples [13, 23, 31]. These combinations are generated during the match procedure as intermediate results and as production instantiations. The total number of such combinations can be a high order polynomial function of the number of tuples [23], which leads to a combinatorial explosion as the number of tuples grows. For the sake of efficiency, almost all existing match algorithms that are capable of matching powerful rules maintain some of these combinations as intermediate *match state* [5, 14].

Research efforts at developing match algorithms with better scaling characteristics have focussed either on maintaining less state [5, 16] or on efficiently maintaining the state on secondary storage [4, 26, 33]. These algorithms retain the *tuple-oriented* nature of the traditional algorithms. That is, they match individual tuples, and generate combinations of individual tuples for intermediate results and instantiations.

We take a different approach. We propose a basic shift in the nature of the match process: from tuple-oriented to *collection-oriented*. In collection-oriented match, collections of tuples that match individual conditions are the unit of matching, rather than individual tuples. The intermediate results and instantiations of collection-oriented match are combinations of these collections, instead of combinations of individual tuples. For large amounts of data, the number of tuples that match individual conditions is likely to be large. In such situations, the number of collections will be much smaller than the number of

tuples. This allows collection-oriented match to tame the combinatorial explosion, since it generates combinations of collections instead of collections of tuples. Thus, it promises substantial improvements in space and time requirements over tuple-oriented match algorithms without giving up expressiveness.

Collection-oriented match also addresses another problem with tuple-oriented match algorithms. These algorithms do not provide efficient support for collection-oriented production languages. Collection-oriented production languages are arguably more suitable for integration with relational databases than tuple-oriented languages, since the unit of operation in relational databases is a relation rather than a single tuple. Collection-oriented languages also make it possible to specify aggregate operations like count, sum, statistical operations like mean and variance, data-fitting operations etc which are important for database-based tasks [8, 12, 34]. While such operations *can* be expressed in tuple-oriented languages, they cannot take advantage of optimized procedures for these operations. Collection-oriented match provides direct support for collection-oriented production languages. If needed, it can also be used to implement a tuple-oriented production system such as OPS5 [6].

We have found that many of the tuple-oriented match algorithms can be easily transformed to their collection-oriented analogues. To illustrate this, we describe the transformation of Rete to *Collection Rete*, its collection-oriented analogue. To investigate the efficiency and scalability of Collection Rete, we used it to implement a collection-oriented extension of OPS5, called COPL. We compared the performance of this implementation with that of an optimized Rete-based OPS5 implementation on a set of scalable benchmarks. Results show that, on these benchmarks, a relatively underoptimized implementation of Collection Rete achieves between one and four orders of magnitude improvement in time *and* up to one order of magnitude improvement in space over an optimized Rete implementation.

We believe this work has both scientific and engineering impacts. On the scientific front, these results provide information on the efficiency and scaling characteristics of tuple-oriented and collection-oriented match approaches. From the engineering perspective, we view them as establishing the feasibility of collection-oriented match for matching large amounts of data and, therefore, for its use in integrated database-production systems. In several of the experiments, the Collection Rete implementation (with its limitations) matched over a million tuples within a reasonable time period. To the best of our knowledge, this is approximately two orders of magnitude larger than the largest working memory previously dealt with. Clearly, these results are specific to our benchmarks and more research on collection-oriented match algorithms will be required before such large working memories can be routinely dealt with.

The rest of this paper is organized as follows. Section 2 introduces OPS5 and the terminology that we will use in the rest of the paper. Section 3 presents collection-oriented match, and describes how it supports collection-oriented production languages. Section 4 describes Rete, and its transformation into Collection Rete. Section 5 describes our benchmarks, experimental methodology, results and analyses. Section 6 addresses the issue of validity of these results. Section 7 discusses related work and the implication of the collection-oriented approach for other match algorithms. Finally, Section 8 presents conclusions and discusses issues for future work.

2. Background

Various languages have been proposed for integrated database-production systems. Many researchers have focused on using OPS5 or OPS5-style production system languages for this integration [5, 8, 12, 16, 27]. This section introduces OPS5 terminology using the simple production system in Figure 2-1.

Figure 2-1-a shows the working memory in the system, essentially a relational database. The working memory contains nine tuples (or working memory elements): W1,W2...W9. The symbols GOAL and

EMPLOYEE are called the classes of the tuples, and correspond to relations. The up-arrows (^) in the tuples indicate attribute names, and correspond to the fields in a relation. These tuples are to be matched with the production MAKE-TEAM, shown in Figure 2-1-b. The production has three conditions on its condition-side or LHS, and one action on its action-side or RHS. The symbols in the conditions are either constants, e.g., HARDWARE, that test if these constants appear in specific fields of the tuples, or variables (enclosed in <>) that bind to values appearing in identical fields in the tuples. The production MAKE-TEAMS teams up a pair of employees, who have worked together on a previous project, but have different areas of expertise. The *make* command on the action side actually creates a new tuple of class TEAM that includes the two members.

```
W1: (GOAL  ^TYPE CREATE-TEAM)

W2: (EMPLOYEE  ^NAME A  ^PREVIOUS-PROJECT WARP  ^EXPERTISE HARDWARE)
W3: (EMPLOYEE  ^NAME B  ^PREVIOUS-PROJECT WARP  ^EXPERTISE HARDWARE)
W4: (EMPLOYEE  ^NAME C  ^PREVIOUS-PROJECT PSM  ^EXPERTISE HARDWARE)
W5: (EMPLOYEE  ^NAME D  ^PREVIOUS-PROJECT PSM  ^EXPERTISE HARDWARE)

W6: (EMPLOYEE  ^NAME E  ^PREVIOUS-PROJECT WARP  ^EXPERTISE COMPILERS)
W7: (EMPLOYEE  ^NAME F  ^PREVIOUS-PROJECT WARP  ^EXPERTISE COMPILERS)
W8: (EMPLOYEE  ^NAME G  ^PREVIOUS-PROJECT PSM  ^EXPERTISE COMPILERS)
W9: (EMPLOYEE  ^NAME H  ^PREVIOUS-PROJECT PSM  ^EXPERTISE COMPILERS)
```

(a)

```
(PRODUCTION  MAKE-TEAM
    (GOAL  ^NAME CREATE-TEAM)
    (EMPLOYEE  ^NAME <N1>  ^PREVIOUS-PROJECT <P>  ^EXPERTISE HARDWARE)
    (EMPLOYEE  ^NAME <N2>  ^PREVIOUS-PROJECT <P>  ^EXPERTISE COMPILERS)
 -->
    (MAKE  TEAM  ^FIRST-MEMBER <N1>  ^SECOND-MEMBER <N2>))
```

(b)

Figure 2-1: A simple production system: (a) Working memory, and (b) A production.

Tuple-oriented match in a production system involves finding all possible tuple-oriented instantiations of the production given the tuples. A tuple-oriented instantiation is a combination of tuples that provide consistent bindings for the variables in the production. In Figure 2-1-a, the instantiation (W1, W2, W6) provides one such consistent binding WARP for the variable <P>. Seven other production instantiations are also generated: (W1, W3, W6), (W1, W2, W7), (W1, W3, W7), (W1, W4, W8), (W1, W5, W8), (W1, W4, W9), and (W1, W5, W9). When instantiations *fire*, action side is executed in the context of its variable bindings, updating the working memory of the system.

3. Collection-Oriented Match

Collection-oriented match treats collections of tuples, rather than individual tuples as the primary objects to be matched. A collection-oriented match algorithm matches each condition in the productions with a collection of tuples and generates *collection-oriented instantiations*, instantiations that have collection of tuples corresponding to each condition in the production. All tuples in the collections are *guaranteed to be mutually consistent*. The following example clarifies this point.

Consider the production system shown in Figure 2-1. Here, collection-oriented match results in two collection-oriented instantiations. The first is ({W1}, {W2,W3}, {W6,W7}). Here, {W1} matches the first condition, {W2, W3} matches the second condition and {W6, W7} matches the third condition. The tuples in these three collections are mutually consistent with each other, i.e., they have consistent values

for the variable <P> — W1 is consistent with W6 and W7, W2 is consistent with W6 and W7, and so on. Similarly, the second collection-oriented instantiation is ({W1}, {W4,W5}, {W8,W9}).

A comparison of these collection-oriented instantiations with the tuple-oriented instantiations presented earlier illustrates two useful points. First, the two types of instantiations contain identical information about consistency of matching tuples. Thus, the tuple-oriented instantiations can be easily generated from collection-oriented instantiations by creating a cross product of its component collections. For instance, a cross product of {W1}, {W2,W3}, {W6,W7}, generates the first four tuple-oriented instantiations.

Second, the comparison illustrates the source of execution space and time efficiency in collection-oriented match. As mentioned earlier, the primary cause of high space and time costs in tuple-oriented algorithms is the generation of a large number of combinations of individual tuples. Collection-oriented match cuts down on the number of such combinations. For instance, suppose each of the three collections in one of the collection-oriented instantiations above contained N elements — the instantiation would consume (N + N + N =) O(N) space. On the other hand, tuple-oriented match would create a cross product of ($N \times N \times N$) tuple combinations as its instantiations; and consume $O(N^3)$ space. If these instantiation are maintained as part of the match state, as they are in many match algorithms, then the space savings from collection-oriented match will be $O(N^3)$. For a production with K conditions, the asymptotic savings are $O(N^K)$. Actual savings are even greater for algorithms that maintain intermediate products — since the constant factor is larger for them.

A reduction in the number of combinations, either intermediate products or instantiations, leads to corresponding speedups in execution time. For instance, avoiding the generation of $O(N^K)$ instantiations will lead to correspondingly large speedups. Furthermore, the overheads of updating and maintaining combinations as part of the match state are also reduced dramatically.

The preceding arguments establish two factors as influencing the speedup and match state reduction in collection-oriented match: (i) the size of the collections that match individual conditions (N in the preceding paragraph), and (ii) the number of conditions in the productions (K in the preceding paragraph). A third such factor is the amount of *fragmentation* in the component collections. The example in Figure 2-1 shows a simple case of fragmentation. Suppose all of the EMPLOYEE tuples in the figure had an identical value for the PREVIOUS-PROJECT field, say MACH. Then matching the production MAKE-TEAM using collection-oriented match would have resulted in a single collection-oriented instantiation ({W1}, {W2,W3,W4,W5}, {W6,W7,W8,W9}). However, the EMPLOYEE tuples in Figure 2-1-a have two different values for the PREVIOUS-PROJECT field. Therefore, two different collection-oriented instantiations, with smaller component collections, are formed. It is as though the collection-oriented instantiation with the larger collections has fragmented. In general, fragmentation may lead to the formation of many collection-oriented instantiations with smaller component collections. This increases both space and time requirements since more match state has to be generated and maintained. However, fragmentation occurs because of the necessity to maintain consistency between collections. It is thus a feature of the program and not the implementation.

Collection-oriented match does not improve the worst-case space and time complexity of production match. It is still possible to encode NP-complete problems such as hamiltonian circuit or subgraph isomorphism within the match of a single production.

3.1. Collection-oriented languages

As discussed earlier, tuple-oriented instantiations can be easily generated, as needed, from the collection-oriented instantiations. This allows collection-oriented match to function simply as an efficient match implementation for tuple-oriented production systems such as OPS5.

More importantly, collection-oriented match provides direct and efficient implementation support for collection-oriented production languages [12, 34]. At the core of such languages is the capability to directly manipulate collections (or sets) as single entities, instead of manipulating them on an element by element basis. These languages directly support collection-oriented operations such as counting, mean, variance etc. In collection-oriented match, a single instantiation packages together collections that are consistent with each other. This allows the action side of a production to be executed in the context of values from a collection of tuples, rather than values from an individual tuple. For instance, in Figure 3-1, <EMP> will be bound to the entire collection of EMPLOYEEs with an expertise in COMPILERS. Thus, given the working memory from Figure 2-1-b, <EMP> will be bound to {W6,W7,W8,W9}. The function CARDINALITY on the action side then counts these employees, and creates the tuple (COMPILER-EXPERTS ^COUNT 4).

```
(PRODUCTION  COUNT-COMPILER-EXPERTS
     (GOAL  ^TYPE COUNT-COMPILER-EXPERTS)
   ( (EMPLOYEE  ^NAME <X>  ^EXPERTISE COMPILERS) <EMP> )
  -->
     (MAKE COMPILER-EXPERTS  ^COUNT (CARDINALITY <EMP>) ))
```

Figure 3-1: A simple example of a collection-oriented action.

Another language-related issue is the role of negated conditions in production systems. Negated conditions are commonly used to restrict the match and sequence the firing of instantiations. In many cases, the collection-oriented approach obviates such usage. A example is iterating over a collection of tuples. Figure 3-2 shows how the counting operation performed by the collection-oriented production in Figure 3-1 would be done in a tuple-oriented language.

```
(PRODUCTION  COUNT-COMPILER-EXPERTS
     (GOAL  ^TYPE COUNT-COMPILER-EXPERTS)
     ( (EMPLOYEE  ^NAME <X>  ^EXPERTISE COMPILERS)  <EMP> )
     ( (COMPILER-EXPERTS  ^COUNT <VAL>)  <C> )
     - (EMPLOYEE  ^NAME <X>  ^EXPERTISE COMPILERS  ^COUNTED YES)
  -->
     (MODIFY <C>  ^COUNT (COMPUTE <VAL> + 1))
     (MODIFY <EMP>  ^COUNTED YES))
```

Figure 3-2: Counting in a tuple-oriented production system.

We have developed a language called COPL (Collection-Oriented Production Language) with collection-oriented semantics. COPL extends OPS5 in three ways. First, the instantiations of COPL productions are collection-oriented, i.e., the variables are bound to collections of values instead of individual values. Second, COPL actions are collection-oriented. For instance, the *make* action creates a collection of tuples instead of a single tuple, the *remove* action removes a collection of tuples, and so on. Third, as part of its actions, COPL supports calls to functions that operate on collections of values. These functions perform collection-oriented operations (such as count, sum, etc.) or element-wise operations, and return collections of values. Work on COPL is currently in preliminary stages, and many issues need to be resolved. Nonetheless, we have used this version of COPL for our experiments.

4. Transforming a Match Algorithm

To obtain a concrete basis for investigating collection-oriented match, we elected to transform the Rete match algorithm [10] to its collection-oriented analogue: Collection Rete. The decision to transform Rete was based on three reasons. First, Rete is the most commonly used algorithm in production system implementations. Second, Rete is intended for systems with a relatively slow rate of change of tuples. Database systems are expected to have a slow rate of change. Third, an implementation of the Rete algorithm was available to us. (As Section 7 shows, other algorithms can be similarly transformed.) In order to understand Collection-Rete, it is first useful to understand Rete itself. Section 4.1 describes Rete; subsequently, Section 4.2 will describe Collection Rete.

4.1. Rete

Rete employs two main optimizations: (i) it maintains match state from previous computations and (ii) it shares common parts of conditions in a single production or across productions to reduce match effort. We will use the simple production system shown in Figure 2-1 to explain Rete. In Figure 2-1, tuples W1,W2...W9 are to be matched with the production MAKE-TEAM. Rete's operation in matching this production can be understood using the analogy of water-flow through pipes. As shown in the upper part of Figure 4-1-a, each condition of the production can be considered as a pipe, ending in a bucket. The tuples flow through these pipes. Each pipe has filters associated with it, which correspond to the constant tests in the condition, and allow only particular tuples to pass through. For example, the filters in the first pipe (corresponding to the first condition) check if the tuple is of class GOAL, and has the value CREATE-TEAM for its TYPE field. Therefore, only W1 passes through the first pipe and appears in the bucket for the first pipe. Note that since the filter EMPLOYEE for the second and third pipes tests an identical field of the tuples, this filter is shared between the two pipes, illustrating the sharing optimization.

Next, the small boxes with X_s inside them check for consistency between tuples. Since there is no variable test between the first two conditions, the box that joins the first two conditions does not perform any consistency checks. The tuple W1 is therefore found consistent with each of the tuples W2,W3,W4, and W5. This leads to the creation of the four tuple combinations: (W1,W2), (W1,W3), (W1,W4) and (W1,W5), which are stored in the following bucket. Now the second small box checks the consistency of each of these four combinations against the tuples matching the third condition: W6, W7, W8 and W9. This involves testing the consistency of bindings for variable <P>. Eight different combinations of tuples (W1,W2,W6),...(W1,W5,W9) succeed and form instantiations of this production. These instantiations are stored in an *instantiation set.*

The buckets and the instantiation set in Figure 4-1-b contain the match state of Rete. If a new tuple, say W10, is added to the system, then only W10 will be matched; Rete will avoid re-matching W1,W2...,W9. The penalty for maintaining this state is that if a tuple is deleted, it has to be deleted from all the combinations that contain it.

In a Rete implementation, the productions are compiled to a dataflow network as shown in Figure 4-1-b. Tuples travel down from the ROOT node. The filters that test for constants are called constant test nodes. The buckets that store individual tuples are called alpha memories. The tuples stored in alpha memories are called *right tokens*. Combinations of tuples, e.g., (W1, W4), are called *left tokens* and are stored in beta memories. And-nodes perform consistency-checks, while P-nodes add (and delete) instantiations to the instantiation set. The tuple combinations mentioned as the cause of the poor scalability of tuple-oriented algorithms are the left tokens and the instantiations.

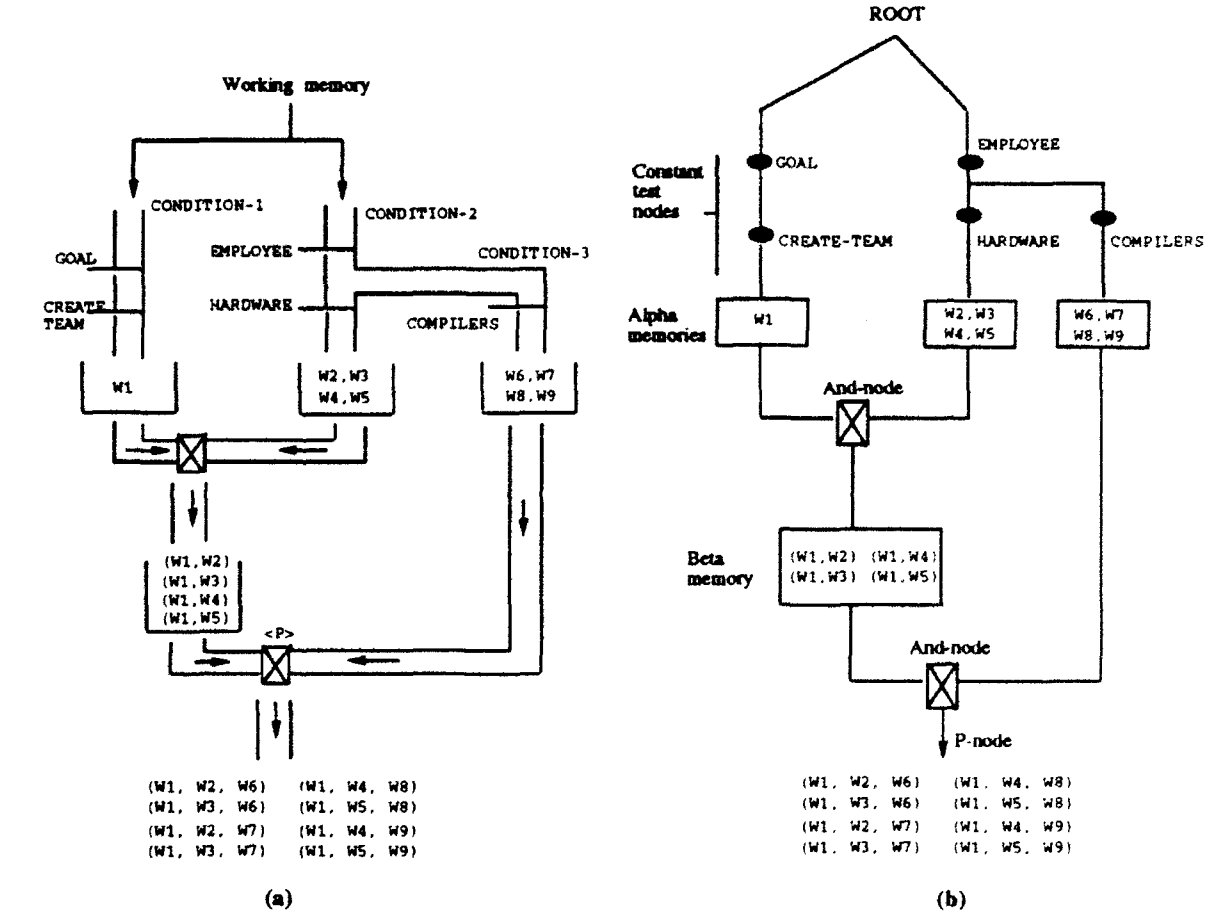


Figure 4-1: Rete algorithm: (a) Analogy of water-flow through pipes, and (b) Dataflow network.

4.2. Collection Rete

We will use the example in Figure 2-1 to describe Collection Rete. Figure 4-2-a shows the transformation of the Rete from Figure 4-1 into Collection Rete. (The detailed algorithm appears in Appendix I.)

In Collection Rete, each condition matches a collection of tuples. Therefore, alpha memories in Collection Rete store collections of tuples that match particular conditions. The first alpha memory stores {W1}. Since there is no consistency test between the first two conditions, {W1} is found consistent with the collection {W2,W3,W4,W5} from the second alpha memory, forming a single *left collection-token* ({W1}, {W2,W3,W4,W5}). In forming such a token, two constraints are observed. First, all the tuples in the component collections of the token are guaranteed to be mutually consistent. Second, to obtain maximum benefit from collection-oriented match, largest possible left collection-tokens are formed. For instance, two separate left collection-tokens ({W1}, {W2,W3}) and ({W1}, {W4,W5}) could potentially be formed instead of the single ({W1}, {W2,W3,W4,W5}), but this fragmentation has been avoided.

At this point, the left collection-token ({W1}, {W2,W3,W4,W5}) formed is stored in the beta memory as shown. It is then checked for consistency with {W6,W7,W8,W9} in the third alpha memory to form new tokens. Given the consistency test for variable <P> at this point, forming left collection-tokens

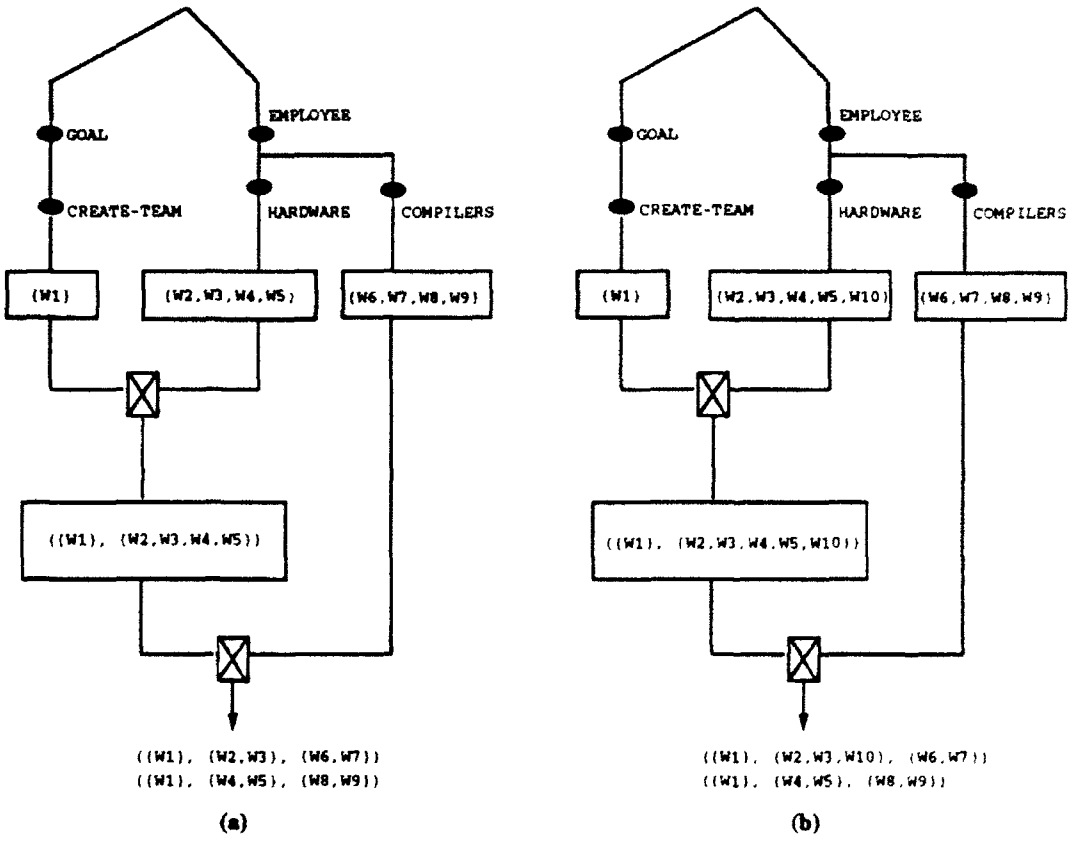


Figure 4-2: Collection Rete: (a) Transformation of Figure 4-1, and (b) Adding a tuple W10.

becomes more complex. A simple method for forming such tokens is as follows. The left collection-token is first sequentially compared with each element of the collection in the third alpha memory — W6,W7,W8, and W9. During this process, a subpart ({W1}, {W2,W3}) of the left collection-token is found consistent with the tuple W6, forming a new left collection-token ({W1}, {W2,W3}, {W6}). The subpart is also found consistent with the tuple W7, forming ({W1}, {W2,W3}, {W7}). Two other new left collection tokens are also formed: ({W1}, {W4,W5}, {W8}) and ({W1}, {W4,W5}, {W9}).

While these four newly formed left collection-tokens are consistent, they do not contain the largest collections possible. For this, a *merging* step is required. This step maintains the consistency of component collections. For two tokens to be merged, they must differ in exactly one component. For instance, ({W1}, {W2,W3}, {W6}) and ({W1}, {W2,W3}, {W7}) can be merged together to form ({W1}, {W2,W3}, {W6,W7}). Similarly, the remaining two tokens may be merged to form: ({W1}, {W4,W5}, {W8,W9}). After this, no further merging can take place. Since this is the final condition of the production, the two tokens ({W1}, {W2,W3}, {W6,W7}) and ({W1}, {W4,W5}, {W8,W9}) are sent to the P-node as collection-oriented instantiations.

Figure 4-2-b shows the operation of Collection Rete when a new tuple W10 of the form (EMPLOYEE ^NAME I ^PREVIOUS-PROJECT WARP ^EXPERTISE HARDWARE) is added to the working memory. W10 becomes a member of the collection matching the second alpha memory. As in Rete, W10 is then checked for consistency with the contents of the previous alpha memory. Since there is no

consistency test, a left collection-token ({W1}, {W10}) is formed. This token is then merged with the token in the following beta memory to form ({W1}, {W2,W3,W4,W5,W10}). While ({W1}, {W10}) is merged, it is also compared with the collection in the third alpha memory. The token ({W1}, {W10}, {W6,W7}) results from this comparison. This token is merged with an existing collection-oriented instantiation that differs in only one slot, giving rise to the instantiation ({W1}, {W2,W3,W10}, {W6,W7}) as shown.

If a tuple is deleted, then it follows a course symmetrical to its addition. Suppose W10 is deleted from the Collection Rete in Figure 4-2-b. W10 is first deleted from the collection in the second alpha memory. A new left collection-token ({W1}, {W10}) is formed, with a *delete* flag. This token is then propagated to the following memory nodes. This token causes *breaching* of the tokens in the succeeding beta memories and in the instantiation set. Breaching undoes the effect of merging. The final result of this processing is that the Collection Rete in Figure 4-2-b is reverts to its state in Figure 4-2-a.

This is the basic form of Collection Rete. A battery of optimizations can be applied to this basic structure. Some of these optimizations are simply transplanted from Rete. An example of this is the *delete optimization* introduced by Scales [25]. When a tuple such as W10 is deleted from an alpha memory, new left collection-tokens are not formed; instead, the successor beta memories (and the instantiation set) are scanned, and any copy of the tuple in any token and instantiation is eliminated. This automatically achieves breaching. Several other optimizations, some targeted specifically towards the merging process — potentially a very expensive step in Collection Rete — are also possible. For instance, a collection in an alpha memory can be pre-emptively divided into two or more *equivalence classes*, where tuples within a single equivalence class have identical values for the fields tested for consistency. In Figure 4-2, {W6,W7,W8,W9} can be divided into two equivalence classes: {W6,W7} and {W8,W9}. Tuples in these two classes have identical values for the field PREVIOUS-PROJECT. Thus, if the first tuple in a class is found consistent, then the entire class is guaranteed to be consistent, which greatly reduces the merging effort for the collection.

Hashing alpha and beta memories achieves a significant speedup for the tuple-oriented Rete algorithm [15]. Hashing quickly isolates the tokens in memories that are likely to match. In Collection Rete, however, the gains from hashing are not expected to be as high. Each left collection-token corresponds to many tuple-oriented tokens. As a result, much less effort is needed to isolate matching tokens. Furthermore, hashing involves additional overheads, since the left collection-token has to be placed in multiple buckets corresponding to the number of tuples in the collections being tested. These considerations do not hold for alpha memories and they could still be hashed. However, the equivalence-classes optimization already provides some of the advantages of hashing by partitioning the list of tokens.

5. Experiments

5.1. Benchmarks

We benchmarked implementations of Collection Rete and an optimized version of Rete. The Collection Rete implementation was done as a part of a COPL implementation. For a Rete implementation, we used *CParaOPS5* [1, 18], the public-domain C-based OPS5 implementation available from Carnegie Mellon. It is one of the fastest implementations of Rete and is faster than the previous version whose performance was shown to be comparable to that of *ops5c* developed at University of Texas, Austin [22].

For the implementation of COPL, we modified a derivative of CParaOPS5 to use the Collection Rete algorithm. The COPL implementation uses basic Collection Rete augmented by the delete and equivalence-class optimizations described in in the previous section. We have devised several other optimizations, but have deferred their incorporation till we can evaluate their relative efficacy. Following the argument from the previous section, we have not incorporated memory hashing into our

implementation. The current implementation of COPL does not support negated conditions; work on implementing them is currently in progress. However, as discussed in Section 3.1, COPL obviates the need for many negated conditions. For each of the benchmarks, the CParaOPS5 programs required negated conditions but COPL versions did not.

The benchmark suite consists of three programs that are able to process varying amounts of data. This allows us to investigate efficiency and scalability of the two algorithms. The programs are:

- *make-teams*: This program operates on a database of employees which contains information about their present department, previous project and an evaluation of how good they are. The task is to build teams of employees given constraints that each member must be from a different department and some of the members must have worked together previously. The programs builds and counts teams that are "good", goodness being defined in terms of the individual evaluations. Data for this benchmark was generated by taking the number of employees as an argument and randomly assigning employees to departments and projects.
- *clusters*: This program operates on a collection of image objects that are characterized by their position and type (e.g. road, hangar, tarmac etc.). It computes the distance between the objects, builds clusters and computes their average size. This task is similar to the those performed by some knowledge-based image understanding systems (like SPAM [20]). Data for this benchmark was generated by taking the number of objects as an argument and placing them randomly in a 100x100 grid.
- *airline-route*: This program operates on a database of airline routes which contains information about source, destination and cost of each flight. The task is to find a minimum cost route for a particular traveler given the desired number of stages. If no route with the desired number of flights can be found, it finds the best alternate route. Data for this benchmark was generated by assuming ten airlines and twenty airports. Each airline was randomly assigned a hub, and all flights were routed for that airline as outbound-inbound pairs to random destinations. The cost for each flight is randomly assigned and is the same in both directions.

For each benchmark, we ran a sequence of experiments with progressively larger amounts of data. Each sequence of experiments was continued till the CParaOPS5 version ran up against time and/or space limits. In each case, we extended the series of experiments for the COPL version till it too ran up against similar limits. We gathered numbers on the space and the time requirements and the size of working memory for each experiment. For time measurements, we performed each experiment three times and used the average time. All experiments were performed on a pair of Decstation 5000/200 machines, running Mach 2.5, with 96M memory. All the C code was compiled by the MIPS *cc* compiler with the -O option.[1] The CParaOPS5 compiler had all the optimization switches turned on. To determine the total execution time, we used the */bin/time* facility available in Unix.

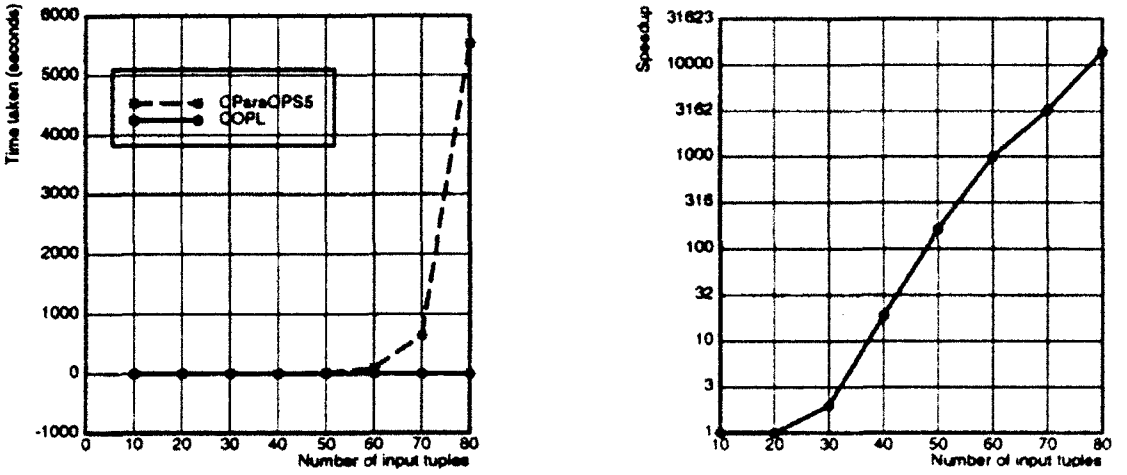
5.2. Results

For each benchmark, we plotted both time and space requirements against the input data size. Note that some of execution-time graphs, the line for COPL programs is slightly above the x-axis, and due to the scaling, appears to lie on it. Graphs for execution time and space show results only for the experiments for which both CParaOPS5 and COPL programs could be run.

Figures 5-1, 5-2 and 5-3 contain the graphs plotting the total time and speedups against the input data size for *make-teams*, *clusters* and *airline-route* respectively. The speedup graphs are plotted on a log scale

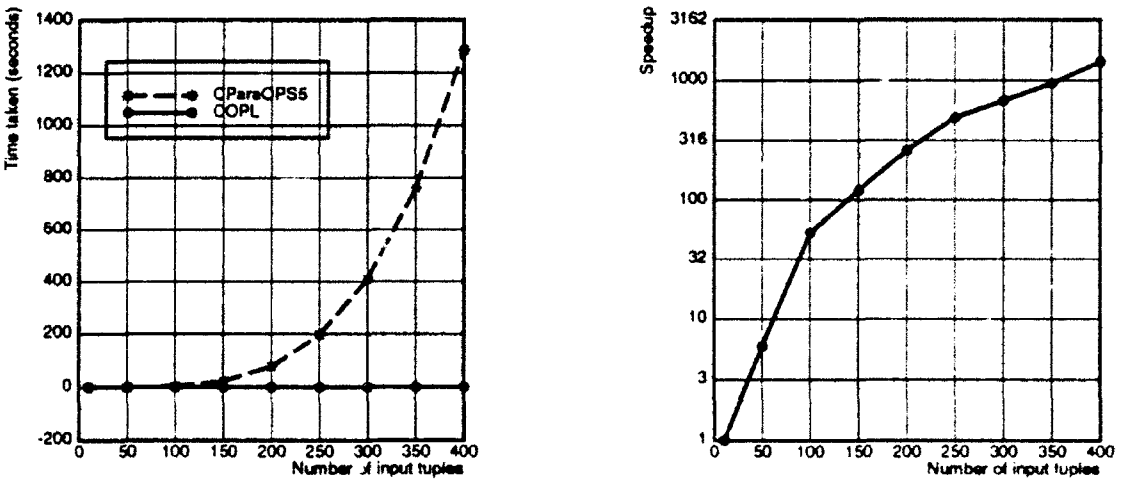
[1]The C code includes COPL and CParaOPS5 libraries, generated code from both the compilers and code for the external functions called.

to accommodate the large range. The inflections in the speedup curves are data dependent and not an artifact of the match algorithm. They are caused by the order in which the random data is generated. The maximum speedups were 13842 for *make-teams*, 1429 for *clusters* and 57 for *airline-route*.



(a) Total time (b) Speedups for COPL

Figure 5-1: Execution time for *make-teams*



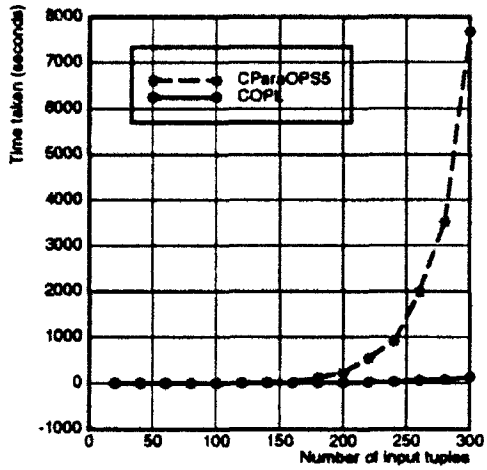
(a) Total time (b) Speedups for COPL

Figure 5-2: Execution time for *clusters*

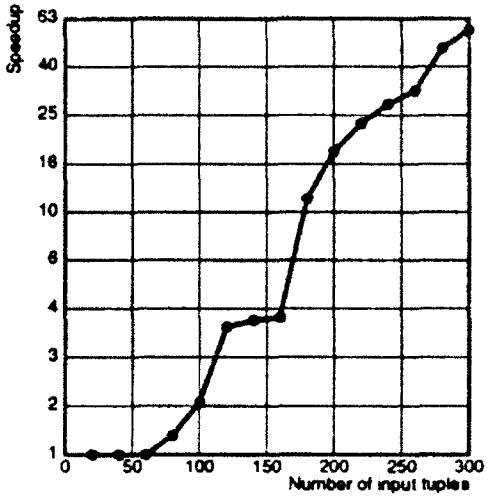
Figures 5-4 and 5-5 contain corresponding graphs plotting the maximum size of the match state (including alpha-memory, beta-memory and the instantiation-set) against the input data size. The highest ratios in the size of match state were 13.6 for *make-teams*, 3.4 for *clusters* and 1.6 for *airline-route*.

In all the three cases, for small amounts of data, the execution time and the size of match state for both CParaOPS5 and COPL are comparable. As the size of input data increases, *COPL soon dominates CParaOPS5*.

Another important metric for production system performance, besides speedup and state reduction, is the maximum size of working memory processed. This is different from the size of the input data and includes all intermediate data generated during the computation. Figures 5-6 and 5-7 show graphs plotting the maximum size of working memory against the execution time. These graphs have been

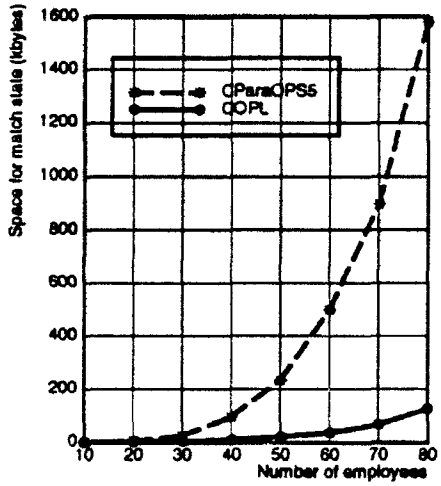


(a) Total time

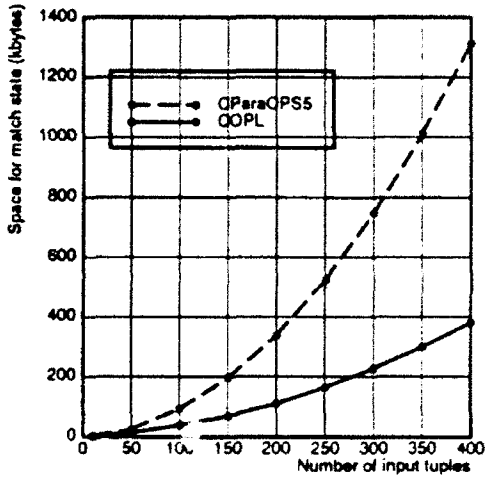


(b) Speedups for COPL

Figure 5-3: Execution time for *airline-route*



(a) *make-teams*



(b) *clusters*

Figure 5-4: Size of the total match state for *make-teams* and *clusters*

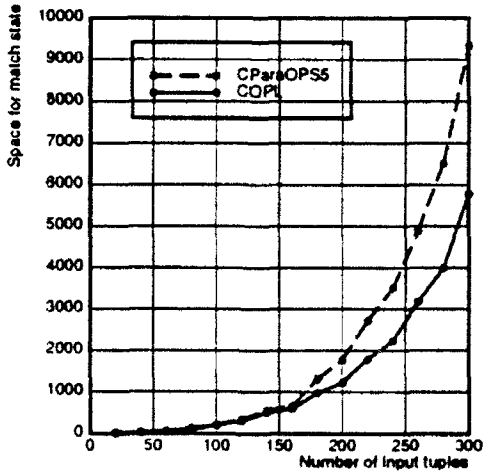
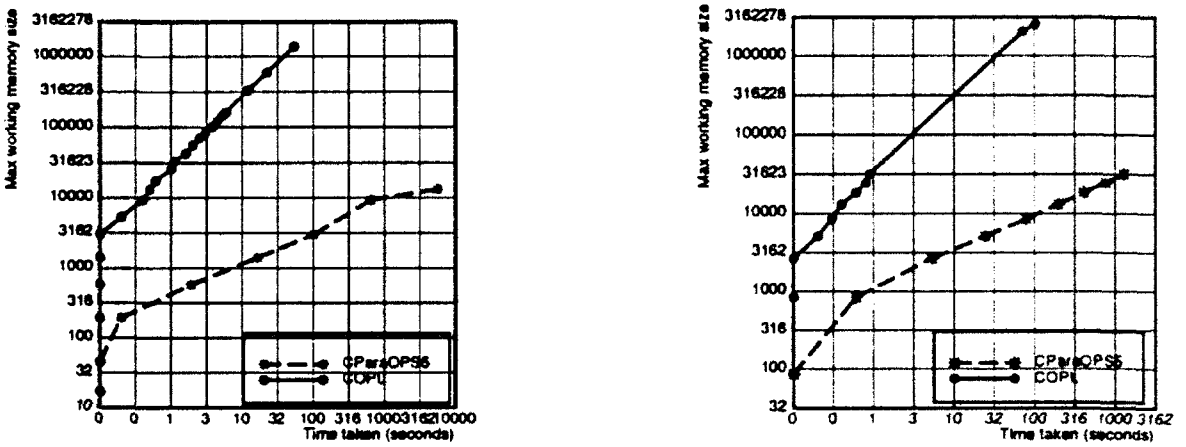


Figure 5-5: Size of the total match state for *airline-route*

plotted on a log-log scale to accommodate the large range on both axes. They allow us to compare the maximum size of working memory that each pair of programs can handle in a given amount of time. These graphs show that in the same amount of time, COPL is able to process up to two orders of magnitude more tuples than CParaOPS5. Furthermore, as more time becomes available, COPL is able to make better use of it than CParaOPS5. These graphs show results for all COPL runs not just those for for which the corresponding CParaOPS5 runs could be completed. The largest working memory processed by COPL programs contained about 1.4 million tuples for *make-teams*, a little over 2.5 million tuples for *clusters* and a little over 200,000 tuples for *airline-route*. Corresponding numbers for CParaOPS5 are 3015, 31813 and 36455. While these numbers do seem small compared to the corresponding COPL numbers, it is important to remember that production system programs have usually dealt with no more than a few thousand tuples.



(a) *make-teams* (b) *clusters*

Figure 5-6: Maximum size of working memory for *make-teams* and *clusters*

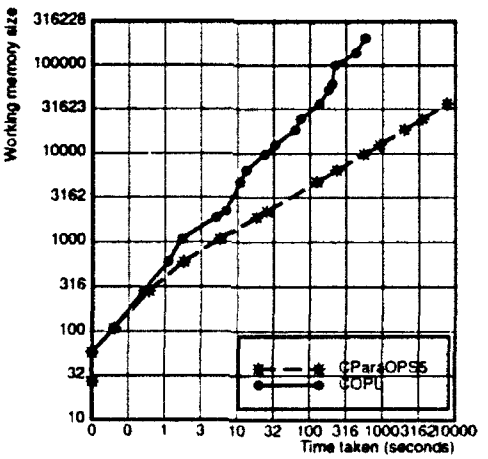


Figure 5-7: Maximum size of working memory for *airline-route*

Raw data for the experiments is available in Appendix II.

5.3. Analysis

As mentioned in Section 3, three factors govern the speedup and the match state reduction achieved by Collection Rete: (i) the size of the collections that match individual conditions, (ii) the fragmentation caused by the constraints between the conditions, (iii) the number of conditions in the productions. Programs that have large productions with large collections matching each condition and little fragmentation will achieve large time and space improvements. Programs that don't have some or all of these characteristics will achieve lower or no improvements. The three benchmarks provide an illustration. All three of them have large collections matching individual conditions, and hence show some performance improvement. All three have some fragmentation, but *airline-route* has a significantly higher amount of fragmentation, and shows correspondingly smaller speedups. Of the other two, *make-teams* has more fragmentation than *clusters*, but it also has more conditions (three-to-four) per production than *clusters* (two conditions). This more than offsets the effect of fragmentation in *make-teams*, and leads to large speedups and match state reduction.

While all three programs achieve match state reduction, this reduction is not as high as the speedups achieved. Partly, this is because alpha memories, which typically consume very little match time, consume a relatively large proportion of match-state. Thus, for beta memories alone, the reduction in match state is much higher. However, there are also some small match-state overheads associated with COPL. First, the COPL implementation imposes additional organization in form of equivalence classes in alpha memories. Second, it introduces additional organization on the left collection-tokens and the collection-oriented instantiations. In particular, while tuple-oriented tokens consist of an array of pointers to tuples, collection-oriented tokens are represented by an array of pointers to collections of tuples, each collection being implemented as a *cons-list*. The extra space consumed by these overheads depends on the amount of fragmentation in the system. For instance, if there is high fragmentation, as in *airline-route*, then the cons-list becomes a factor in the space consumed.

We believe this work has both scientific and engineering impacts. On the scientific front, these results provide information on the efficiency and scaling characteristics of tuple-oriented and collection-oriented match approachs. From the engineering perspective, we view them as establishing the feasibility of collection-oriented match for matching large amounts of data. For the *clusters* benchmarks, our Collection Rete implementation (with its limitations) is able to match over 2.5 million tuples in about 100 seconds; for the *make-teams* benchmark, it is able to match 1.4 million tuples in about 52 seconds. To the best of our knowledge, this is approximately two orders of magnitude larger than the largest working memory previously dealt with. Obviously, these results are specific to our benchmarks and more research will be required before such large working memories can be routinely dealt with.

6. Discussion

Results of the magnitude described in this paper inevitably raise questions about the fairness of the comparison and the validity of these results for real-world applications. By building a highly optimized system and comparing it with a suboptimal system using benchmarks that contain large and unrealistically complex productions, results can be made to look good. However, in our comparison, the *exact opposite* is true. The target of our comparison, CParaOPS5, is actually a highly optimized system, as testified by the following: (i) CParaOPS5 is based on a well-known compilation technique [11] and includes a variety of optimizations (e.g. hashing of memories, caching of global values, aggressive inlining); (ii) An earlier, slower version of CParaOPS5 was found to be only a factor of 1.5 to 2 slower than the optimized *ops5c* system distributed by the University of Texas, Austin [22]; (iii) For small working memory sizes, CParaOPS5's performance is comparable to that of COPL, indicating that COPL does not have some low level implementation advantage; (iv) CParaOPS5 has provided efficient support for the SPAM knowledge-based image recognition system [20], which regularly takes fifty thousand to a

million production firings to run. Compared to CParaOPS5, the COPL system is in a relatively underoptimized state. This is our first implementation of COPL, with no time spent on tuning its performance. Additionally, our benchmark set contains simple productions of three-to-four conditions, with the *clusters* benchmark containing only two conditions per productions. (A larger number of conditions will give COPL much higher speedups.)

Furthermore, traditionally, performance improvements in the production systems world have been confined to single digits [22, 24, 32]. The speedups here go much beyond these single digits. It is against this background that we find the results presented in this paper promising.

However, our benchmarks not completely bias-free. Given that we targeted production systems that will operate with large amounts of data, the benchmarks are dominated by matching of large collections. Will *real* integrated database-production systems show improvements similar to these benchmarks? If these *real* systems match large collections, then they will obtain similar speedups. The size of collections depends on the *selectivity* of the tests and the number of tuples tested. (Selectivity is defined as the percentage of tests, constant or variable, that fail.) Large collections can occur if the selectivity is low, or if the number of tuples tested is high or both. The expensive learned rules in the Soar production system [19] provide one example of low selectivity. These learned rules so expensive to match that they cause Soar to slowdown with learning rather than speeding up [30]. Image recognition systems like SPAM, and other database systems provide examples of systems where the number of tuples tested are high. We expect that as the amount of data processed by production systems grows, the size of collections will grow. Even for systems with high selectivity, large enough amounts of data will lead to large collections.

One important point here is that collection-oriented match supports a collection-oriented programming model. Production match operations, which were previously considered extremely expensive, are no longer so. This will allow a change in the programming style and is likely to expand the scope of applications to tasks which have hitherto been considered intractable.

7. Related Work

In this section, we discuss the implications of collection-oriented match for match algorithms other than Rete. We have found it relatively easy to transform tuple-oriented algorithms to their collection-oriented analogues. We also discuss other work related to matching large amounts of data.

Treat [21] is the other major algorithm found in the production systems literature. The key difference between Rete and Treat is that Treat does not maintain beta memories as a part of its match state; it only maintains the alpha memories and the instantiation set (see Figure 4-1). However, the operations it performs to determine the instantiations are similar to those of Rete — it too creates left tokens, compares these tokens with tuples in alpha memories to create new left tokens and so on. Treat can be easily transformed to *Collection Treat*, along the same lines as Collection Rete. In Collection Treat, left collection-tokens are compared with collections in alpha-memories, instead of individual tuples, and new left collection-tokens are formed. Collection Treat enjoys three advantages over Treat: (i) a reduction in the size of the instantiation set, which is the main source of space consumption in Treat [23]; (ii) a reduction in the total execution time, given that collection-oriented tokens and instantiations are formed; and (iii) direct support for collection-oriented semantics.

A'-Treat [16] is a refinement of Treat which replaces alpha memories by virtual alpha-memories, which do not maintain state. Since collection-oriented match does not change the organization of the alpha memories, A'-Treat can be transformed in the same way as Treat and the advantages listed above for Treat would carry over to A'-Treat.

Miranker et al.'s LEAPS [23] is another tuple-oriented match algorithm that reduces the amount of state saved. LEAPS has shown large performance improvements over Treat in both execution time and space. LEAPS achieves this improvement by exploiting OPS5's syntactic selection (conflict-resolution) strategies — it computes only a single dominant instantiation instead of all the instantiations. As a result, however, LEAPS is unable to support languages that do not depend on such syntactic conflict-resolution strategies, such as Soar [19], PPL [2] and others [3]. LEAPS also does not directly support collection-oriented languages. LEAPS can be transformed to match the dominant collection instead of the dominant tuple from each alpha memory. Collection LEAPS would generate collection-oriented instantiations, which would allow it to support collection-oriented languages without giving up the advantages of LEAPS. However, Collection LEAPS would be unable to support the other referred to above.

Another area of related work has been that of collection-oriented (or set-oriented) production languages. Several such languages have been proposed [8, 12, 34]. Collection-oriented match can provide an efficient implementation substrate for these languages. Gordin and Pasik [12] suggest a modification to Rete to support collection-oriented languages. The resulting algorithm merges the tuple-oriented instantiations generated by Rete to generate collection-oriented instantiations. It does not take advantage of the structure of collections to tame the combinatorial explosion.

Collection-oriented match was motivated by our previous work on tokenless match [31, 30]. Hence, there are some similarities between tokenless match and collection-oriented match. However, tokenless match is targeted towards real-time systems and focuses on achieving a polynomially bounded match by limiting expressiveness. In contrast, collection-oriented match imposes no restrictions on expressiveness. As a result, even though it is able to improve production system performance, it does not improve the asymptotic complexity of the match problem. Understanding the relationship between tokenless match and collection-oriented match remains an interesting issue for future work.

8. Conclusions and Future Work

We can now answer the question that was raised at the beginning of this paper. Yes, collection-oriented match algorithms can support a large number of powerful match operations and yet scale well as the amount of data increases. Results presented in this paper showed that in several cases, which we expect to occur in practice, collection-oriented match is able to match OPS5-style productions against large amounts of data in reasonable amounts of time. While these results are based on an implementation of Collection Rete, a collection-oriented analogue of Rete, this paper also discussed how other tuple-oriented match algorithms can be transformed to their collection-oriented analogues. Based on a preliminary analysis, we expect the transformed algorithms to scale better than the tuple-oriented originals.

While these results have demonstrated large time and space improvements for COPL, much remains to be done. Our immediate plans for further research are to complete the design and implementation of COPL. This will allow us, and others in our environment, to build large applications in COPL. This, in turn, will provide us a better understanding of the nature of computation in collection-oriented match and its utility for real-world tasks. We also hope to use these tasks to evaluate the relative efficacy of hashing and merging optimizations. Another investigation we plan to take up in near future is the development of a uniform framework to understand and evaluate the different ways of dealing with collections in production systems including collection-oriented match and tokenless match.

Acknowledgement

We thank Bob Doorenbos, Dave McKeown, Brian Milnes, Dirk Kalp, and Paul Rosenbloom for helpful discussions on this topic. We thank Gary Pelton and Rick Lewis for loaning us their workstations for the experiments reported in this paper.

Appendix I. Detailed Algorithm for Collection Rete

The following describe the simplest version of Collection Rete. Section I.1 describes the algorithm for adding a tuple to positive conditions. Section I.2 describes the addition of a tuple to negated conditions. Section I.3 describes the deletion of a tuple. The notation used is as follows:

$C_1,...,C_i...C_N$	Conditions of the production matched
Alpha(C_i)	Alpha memory for condition C_i
Beta(C_i)	Beta memory storing the results from the match of Alpha(C_i) with Beta(C_{i-1})
LCT, xx_LCT	Left collection-tokens
Counter(LCT)	Counter associated with LCT
W	A Tuple

In Collection Rete, the processing of alpha constant tests does not change. Therefore, the following description is only for the processing in memories and two-input tests.

I.1. Adding a Tuple: Non-negated Conditions

Figure I-1 describes the procedures involved in adding a tuple W to a non-negated condition C_i. Two main procedures are involved: Add_Tuple and Add_Left_Collection_Token. The procedure Add_Tuple corresponds to a *right-activation* in Rete. Step 1 of Add_Tuple adds W to the collection in Alpha(C_i). Step 2 searches the previous beta-memory Beta(C_{i-1}) for a consistent left collection-token LCT. The procedure *check_consistent* returns LCT if W is consistent with LCT. However, W may be consistent with only a sub-part of LCT. That is, if LCT consisted of i-1 collections of the form (collection_1, collection_2,...collection_i-1), then W may be consistent with only sub_LCT, which consists of (sub_collection_1, sub_collection_2,...sub_collection_i-1), where $sub\_collection\_j \subseteq collection\_j$ for $(1 \leq j \leq i-1)$. In such a case, *check_consistent* will return sub_LCT. *check_consistent* returns NULL if at least one of the *sub_collection_j* is NULL. If *check_consistent* returns a non-null value, then a successor left collection-token Succ_LCT is created, using the routine Create_New_Collection_Token.

The procedure Add_Left_Collection_Token is then called. It corresponds to the *left_activation* in Rete. Steps 1 through 3 add a new left collection-token to a beta memory. For this, step 2 searches for an existing left collection-token in the beta-memory with which the new collection-token may be merged. This merging is achieved by procedure *Merge*. It merges two left collection-tokens if they are identical except for one collection. It also deletes the old (or previous) token. If such merging cannot take place, then step 3 simply adds the new left collection-token to the beta memory. Steps 4 and 5 search the alpha memory of the next condition for matching tuples. They merge the resulting left collection-tokens and create successors. Step 6 recursively calls Add_Left_Collection_Token using these these new successor tokens.

I.2. Adding a Tuple: Negated Conditions

Figure I-2 describes the procedures involved in adding a tuple W to a negated condition C_i. Step 1 of Add_Tuple_Negated simply adds W to the collection in Alpha(C_i). Step 2 searches the previous beta-memory for a matching left collection-token that is consistent. If a consistent token is found then its counter is updated. For a negated condition C_i, a counter is associated with each left collection-token in Beta(C_{i-1}) to count the number of tuples from Alpha(C_i) that have successfully matched it. If this counter makes a transition from 0 to 1, then that token is deleted from the following beta memories. It is possible W is consistent with only a sub-part of the left collection-token. In such an event, the token is itself split into multiple parts — consistent and inconsistent parts — with each part maintaining a counter. The counters for the consistent parts are incremented, but the counters for the other parts are kept unchanged. If the counter again makes a transition from 0 to 1, then this consistent part has to be deleted. This

```
Procedure Add_Tuple(W, i)
Begin
1. Add W to the collection in Alpha(C_i);
2. For each left collection-token LCT in Beta(C_i-1)
    Begin
      con_LCT := check_consistent(W, LCT);
      if (con_LCT != NULL)
        Begin
          Succ_LCT := Create_New_Collection_Token(con_LCT, W);
          Call Add_Left_Collection_Token(Succ_LCT, i+1);
        End
    End
End


Procedure Create_New_Collection_Token(LCT, W)
Begin
1. Copy LCT to New_LCT;
2. Copy W as the last collection of New_LCT;
3. Return(New_LCT);
End


Procedure Add_Left_Collection_Token(LCT, i)
Begin
1. Found := False;
2. for each left collection-token old_LCT in  eta(C_i)
      Begin
          if LCT can merge with old_LCT then
               Begin
                    new_LCT := Merge(LCT, old_LCT);
                    Found := True;
               End
      End
3. if not(Found) then add LCT to Beta(C_i);
4. Merge_list := Null;
5. For each tuple W in Alpha(C_i+1)
     Begin
       con_LCT := check_consistent(W, LCT);
       if (con_LCT != NULL)
        Begin
          Succ_LCT := Create_New_Collection_Token(con_LCT, W);
          if Succ_LCT can merge with prev_LCT on Merge_list then
                  Succ_LCT := Merge(Succ_LCT, prev_LCT);
          Push Succ_LCT on the Merge_list;
        End
     End
6. For each Succ_LCT on Merge_list
         Call Add_Left_Collection_Token(Succ_LCT, i+1);
End
```

Figure I-1: Procedure for adding a tuple: non-negated conditions.

deletion is achieved indirectly so as to maintain uniformity within the delete procedure.

The procedure Add_Left_Collection_Token_Negated describes the *the operation in Beta(C_{i-1})*. The key idea here is to preserve the structure of Beta(C_{i-1}) such that each counter in a left-collection token preserves the exact count of the number of tuples matched. For this purpose, the new token LCT is first compared with tuples in Alpha(C_i). Steps 1 through 3 find consistent matching tuples, and update counters. Since only subparts of LCT may be found consistent, the counting step becomes more complex. Step 4 merges the tokens emerging from step 3 into Beta(C_{i-1}). Step 5 accumulates the inconsistent portion of LCT and sets the counters to zero. For the merges in Step 4 and 5, two tokens must not only be identical in all execpt one collection, but they must also have identical values for their counters.

I.3. Deleting a Tuple

Figure I-3 and I-4 describe the procedures involved in deleting a tuple from non-negated and negated conditions. These procedures are anologous to their corresponding addition procedures. The new concept here is breaching, which undoes the effect of merging. For a left collection-token T to breach another, it must differ with T in at most one slot. If it differs in no slot, then the entire token must be deleted. Another point is that during deletion, within a uniprocessor execution environment, no conjugates [13] will be created. That is, if a deletion token arrives at a beta memory, then that beta memory is guaranteed to have a copy of that token, in a merged or non-merged form.

```
Procedure Add_Tuple_Negated(W, i)
Begin
1. Add W to the collection in Alpha(C_i);
2. For each left collection-token LCT in Beta(C_i-1)
     Begin
       con1_LCT := check_consistent(W, LCT);
       con2_LCT := check_inconsistent(W, LCT);
       if (con1_LCT = LCT)
          Begin
             increment counter(LCT);
             if counter changes from 0 to 1 then
                  Call Delete_Left_Collection_Token(LCT, i);
           End
        else
          Begin
           if (con1_LCT = non-null subpart of LCT)
             Begin
               Split LCT into con1_LCT and con2_LCT;
               increment counter(con1_LCT);
               if counter changes from 0 to 1 then
                Begin
                  Call Delete_Left_Collection_Token(LCT, i);
                  Call Add_Left_Collection_Token(con2_LCT, i);
                End
              End
          End
     End
End


Procedure Add_Left_Collection_Token_Negated(LCT, i)
Begin
1. Merge_list := Null;
2. For each tuple W in Alpha(C_i)
     Begin
       con_LCT := consistent(W, LCT);
       if (con_LCT != NULL)
          Begin
             Counter(con_LCT) := 1;
             Push con_LCT on Count_list;
          End

3. if any con1_LCT and con2_LCT on Count_list identical then
        Begin
           Increment Counter(con1_LCT);
           Delete con2_LCT from Count_list;
        End
4. if LCT from Count_list can merge with old_LCT in Beta(C_i-1)
        then new_LCT := Merge(LCT, old_LCT);

5. if subpart sub_LCT of LCT inconsistent with tuples in Alpha(C_i)
     Begin
        Counter(sub_LCT) = 0;
        if sub_LCT can be merged with old_LCT in Beta(C_i-1)
                then new_LCT = Merge(sub_LCT, old_LCT);
        Call Delete_Left_Collection_Token(old_LCT, i);
        Call Add_Left_Collection_Token(new_LCT, i);
     End
End
```

Figure I-2: Procedure for adding a tuple: negated conditions.

```
Procedure Delete_Tuple(W, i)
Begin

1. Delete W from the collection in Alpha(C_i);
2. For each left collection-token LCT in Beta(C_i-1)
     Begin
       con_LCT := consistent(W, LCT);
       if (con_LCT != NULL)
           Begin
               Succ_LCT = Create_New_Collection_Token(con_LCT, W);
               Call Delete_Left_Collection_Token(Succ_LCT, i+1);
            End
     End
End


Procedure Delete_Left_Collection_Token(LCT, i)
Begin
1. for each old_LCT in Beta(C_i)
      Begin
          if LCT can breach old_LCT then
                 New_LCT := Breach(LCT, old_LCT);
      End
2. Merge_list := Null;
3. For each tuple W in Alpha(C_i+1)
     Begin
       con_LCT := consistent(W, LCT);
       if (con_LCT != NULL) then
          Begin
             Succ_LCT = Create_New_Collection_Token(con_LCT, W);
             if LCT can merge with prev_LCT on Merge_list then
                 Succ_LCT := Merge(Succ_LCT, prev_LCT);
              Push Succ_LCT on the Merge_list;
          End
     End
5. For each Succ_LCT on Merge_list
         Call Delete_Left_Collection_Token(Succ_LCT, i+1);
End
```

Figure I-3: Procedure for deleting a tuple: non-negated conditions.

```
Procedure Delete_Tuple_Negated(W, i)
Begin
1. Delete W from the collection in Alpha(C_i);
2. For each LCT in Beta(C_i-1)
     Begin
       con_LCT := consistent(W, LCT);
       if (con_LCT != NULL)
           Begin
              decrement counter(LCT);
              if LCT now merges with old_LCT in Beta(C_i-1)
                 new_LCT := Merge(LCT, old_LCT);
              if counter changes from 1 to 0 then
                Begin
                 Call Delete_Left_Collection_Token(old_LCT, i);
                 Call Add_Left_Collection_Token(new_LCT, i);
                End
            End
     End
End


Procedure Delete_Left_Collection_Token_Negated(LCT, i)
Begin
1. Merge_list := Null;
2. For each tuple W in Alpha(C_i)
     Begin
       con_LCT := consistent(W, LCT);
       if (con_LCT != NULL)
          Begin
             Counter(con_LCT) := 1;
             Push con_LCT on Count_list;
          End
     End
3. if any con1_LCT and con2_LCT on Count_list identical then
        Begin
           Increment Counter(con1_LCT);
           Delete con2_LCT from Count_list;
        End
4. if LCT from Count_list can breach old_LCT from Beta(C_i-1)
        then New_LCT := Breach(LCT, old_LCT);

5. if subpart sub_LCT of LCT inconsistent with tuples in Alpha(C_i)
     Begin
        Counter(sub_LCT) = 0;
        if sub_LCT can breach prev_LCT in Beta(C_i-1)
           then New_LCT := Breach(sub_LCT, prev_LCT);
        Call Delete_Left_Collection_Token(sub_LCT, i);
     End
End
```

Figure I-4: Procedure for deleting a tuple: negated conditions.

Appendix II. Detailed Experimental Results

This sections contains the raw data from the experiments. The time returned by */bin/time* has only one digit after the decimal leading to quantization problems for small values. For computing speedups, we have assumed that the smallest value of time is 0.1s. This provides a lower bound on the actual speedup for the cases in which the value returned is 0.0 seconds. Note that in the following, **K** implies Kbytes. In tables II-3, II-11, II-12 and II-15, all the ratioes are calculated with respect to COPL numbers.[2]

Input data size	total space (K)	match state (K)	number of tuples	time (s)
10	308	2	17	0.0
20	311	3	45	0.0
30	320	5	194	0.0
40	346	11	582	0.0
50	401	22	1419	0.0
60	498	37	3015	0.0
70	653	69	5383	0.2
80	921	127	9413	0.4
90	1163	167	13280	0.5
100	1420	214	17510	0.6
110	2037	384	25931	1.0
120	2473	452	32999	1.1
130	3131	592	42971	1.6
140	4049	825	56153	2.0
150	5129	1123	71190	2.5
160	5888	1265	83062	3.0
170	7089	1500	101629	3.8
180	8600	1819	124554	4.5
190	10015	2098	146412	5.2
200	11146	2334	163640	5.9
250	22109	4481	333167	12.0
300	40239	8943	596015	22.2
350	64308	14834	945595	35.5
400	93875	21113	1393457	53.5

Table II-1: Data from COPL experiments for *make-teams*

[2]For some tables, some entries are missing, as we were unable to measure all the details.

Input data size	total space (K)	match state (K)	number of tuples	time (s)
10	872	2	17	0.0
20	890	5	45	0.0
30	982	27	194	0.2
40	1220	97	582	1.9
50	1734	234	1419	16.3
60	2755	500	3015	100.1
70	4207	898	5383	644.3
80	6635	1580	9413	5536.8

Table II-2: Data from CParaOPS5 experiments for *make-teams*

Input data size	time ratio	match state ratio
10	1.0	0.89
20	1.0	1.63
30	2.0	5.48
40	10.0	8.82
50	163.0	10.73
60	1001.0	13.62
70	3224	13.11
80	13842	12.44

Table II-3: Comparisons for *make-teams*

Input data size	total space (K)	match state (K)	number of tuples	time (s)
10	310	3	85	0.0
20	313	4	177	0.0
30	319	7	339	0.0
40	326	10	551	0.0
50	335	14	838	0.0
60	348	18	1065	0.0
70	357	24	1517	0.0
80	372	30	1969	0.0
90	384	35	2311	0.1
100	397	39	2663	0.1
110	409	43	2970	0.1
120	426	49	3457	0.2
130	440	53	3804	0.2
140	461	61	4436	0.2
150	483	69	5113	0.2
160	501	75	5605	0.2
170	521	81	6127	0.2
180	546	90	6884	0.2
190	575	102	7791	0.3
200	600	110	8508	0.3

Table II-4: Data from COPL experiments for *clusters* on 100x100 grid - I

Input data size	total space (K)	match state (K)	number of tuples	time (s)
210	631	122	9470	0.4
220	663	134	10462	0.4
230	692	144	11264	0.4
240	720	152	12046	0.4
250	755	165	13108	0.4
260	791	178	14195	0.5
270	825	189	15172	0.5
280	857	199	16029	0.5
290	899	215	17321	0.6
300	938	228	18468	0.6
310	981	244	19785	0.6
320	1021	257	20922	0.6
330	1059	268	21994	0.7
340	1104	284	23341	0.7
350	1151	300	24738	0.8
360	1199	317	26165	0.9
370	1242	330	27342	0.9
380	1295	349	28974	0.9
390	1340	363	30231	0.9
400	1393	381	31813	0.9

Table II-5: Data from COPL experiments for *clusters* on 100x100 grid - II

Input data size	total space (K)	match state (K)	number of tuples	time (s)
410	1449	402		0.9
420	1563	420		1.0
430	1558	438		1.0
450	1676	479		1.2
460	1737	500		1.2
470	1802	524		1.2
480	1855	539		1.2
490	1924	564		1.3
500	1984	583		1.3
510	2052	607		1.3
520	2120	630		1.4
530	2194	657		1.4
540	2266	682		1.5
550	2378	707		1.6
560	2415	734		1.6
570	2495	763		1.7
580	2577	793		1.7
590	2659	823		1.8
600	2741	852		1.9

Table II-6: Data from COPL experiments for *clusters* on 100x100 grid - III

Input data size	total space (K)	match state (K)	number of tuples	time (s)
610	2826	883		1.9
620	2893	902		2.0
630	2981	934		2.0
640	3058	957		2.1
650	3140	984		2.1
670	3297	1033		2.2
680	3392	1068		2.3
690	3486	1101		2.3
700	3586	1137		2.4
800	4597	1491		3.0
900	5755	1900		3.8
1000	5867	1936		3.9
1100	8369	2788		5.5
1200	9847	3282		6.5
1300	11528	3869		7.6
1400	13284	4463		8.7
1500	15225	5138		9.9
1600	17335	5896		11.4
1700	19480	6608		12.7
1800	21776	7391		14.2
1900	24183	8206		15.6

Table II-7: Data from COPL experiments for *clusters* on 100x100 grid - IV

Input data size	total space (K)	match state (K)	number of tuples	time (s)
2000	18971	3899		12.8
2200	22870	4705		15.3
2400	27085	5552		18.1
2600	31678	6479		21.0
2800	36658	7489		24.3
3000	41972	8560		27.9
3200	47675	9716		31.4
3400	53727	10934		35.6
3600	60159	12234		39.7
3800	66955	13606		49.1
4000	74125	15057		48.8
4500	93596	18968	2063409	71.9
5000	115352	23336	2546294	102.9

Table II-8: Data from COPL experiments for *clusters* on 400x400 grid

Input data size	total space (K)	match state (K)	number of tuples	time (s)
10	884	3	85	0.0
20	913	6	177	0.0
30	960	1	339	0.2
40	1023	18	551	0.3
50	1104	27	838	0.6
60	1188	36	1065	0.9
70	1308	49	1517	1.7
80	1439	64	1969	2.7
90	1564	78	2311	3.6
100	1701	93	2663	5.3
110	1842	109	2970	7.4
120	2014	129	3457	10.3
130	2178	147	3804	13.0
140	2387	171	4436	18.3
150	2610	196	5113	24.3
160	2820	219	5605	32.3
170	3049	245	6127	40.2
180	3306	274	6884	50.6
190	3594	307	7791	64.3
200	3870	338	8508	79.1

Table II-9: Data from CParaOPS5 experiments for *clusters* on 100x100 grid - I

Input data size	total space (K)	match state (K)	number of tuples	time (s)
210	4184	374	9470	98.5
220	4512	411	10462	118.1
230	4826	446	11264	141.3
240	5148	483	12046	167.9
250	5512	524	13108	197.5
260	5889	567	17195	231.5
270	6262	609	15192	269.0
280	6631	651	16029	311.7
290	7061	700	17321	358.4
300	7483	748	18468	409.8
310	7935	799	19785	467.9
320	8375	849	20922	532.1
330	8817	899	21994	599.6
340	9302	954	23341	675.3
350	9801	1011	24738	760.2
360	10314	1069	26165	847.9
370	10807	1125	27342	942.9
380	11363	1188	28974	1051.5
390	11884	1247	30231	1167.2
400	12453	1312	31813	1286.0

Table II-10: Data from CParaOPS5 experiments for *clusters* - II

Input data size	time ratio	match state ratio
10	1	1.2
20	1	1.3
30	2	1.6
40	3	1.7
50	6	1.9
60	9	2
70	17	2.1
80	27	2.1
90	36	2.3
100	53	2.4
110	74	2.6
120	51.5	2.6
130	65	2.8
140	91.5	2.8
150	121.5	2.8
160	161.5	2.9
170	201	3
180	253	3
190	214	3
200	263.6	3.1

Table II-11: Comparisons for *clusters* on 100x100 grid

Input data size	time ratio	match state ratio
210	246	3.1
220	295	3.1
230	350	3.1
240	419	3.2
250	494	3.2
260	463	3.2
270	538	3.2
280	623	3.3
290	598	3.3
300	683	3.3
310	780	3.3
320	887	3.3
330	857	3.4
340	964	3.4
350	950	3.4
360	942	3.4
370	1048	3.4
380	1168	3.4
390	1297	3.4
400	1429	3.4

Table II-12: Comparisons for *clusters*

Input data size	total space (K)	match state (K)	number of tuples	time (s)
20	323	17	27	0.0
40	347	38	58	0.1
60	374	62	109	0.2
80	450	126	282	0.5
100	552	207	599	1.1
120	683	306	1082	1.7
140	939	507	1892	5.1
160	1049	595	2216	6.9
180	1603	978	4741	10.8
200	1974	1233	6446	13.1
220	2757	1788	9740	23.6
240	3381	2230	12494	32.7
260	4740	3178	18546	62.7
280	5970	3998	24577	73.9
300	8556	5776	36455	135.8
320	11761	7898	52404	177.8
340	13664	9158	61866	200.8
360	21097	13952	100658	220.3
380	29092	19281	139880	426.7
400	41757	2733	203881	574.5

Table II-13: Data from COPL experiments for *airline-route*

Input data size	total space (K)	match state (K)	number of tuples	time (s)
20	882	6	27	0.0
40	913	22	58	0.1
60	963	45	109	0.2
80	1113	103	282	0.6
100	1374	197	599	1.8
120	1783	351	1082	5.7
140	2420	560	1892	18.2
160	2699	668	2216	25.3
180	4673	1309	4741	123.5
200	6029	1765	6445	233.7
220	8976	2717	9740	547.9
240	1127	3514	12494	912.7
260	16050	4885	18506	1984.9
280	211[illegible]	6511	24577	3512.0
300	[illegible]467	9334	36455	7674.8

Table II-14: Data from CParaOPS5 experiments for *airline-route*

Input data size	time ratio	match state ratio
20	1.0	0.4
40	1.0	0.6
60	1.0	0.7
80	1.2	0.8
100	1.6	1.0
120	3.4	1.1
140	3.6	1.1
160	3.7	1.1
180	11.4	1.3
200	17.8	1.4
220	23.2	1.5
240	27.9	1.6
260	31.7	1.5
280	47.8	1.6
300	56.6	1.6

Table II-15: Comparisons for *airline-route*

References

1. Acharya A., and Kalp, D. Release Notes for CParaOPS5 5.3 and ParaOPS5 4.4. Distributed with the CParaOPS5 release available from Carnegie Mellon University.

2. Acharya, A. PPL: An explicitly parallel production language for large scale parallelism. Proceedings of the IEEE conference on Tools for AI, 1992, pp. 473-474.

3. Barachini, F. "The evolution of PAMELA". *Expert Systems 8*, 2 (1991), 87-98.

4. Bein J., King, R., Kamel, N. MOBY: An Architecture for Distributed Expert Database Systems. Proceedings of the Thirteenth International Conference on Very Large Databases, 1987, pp. 13-20.

5. Brant, D. A., Grose, T., Lofaso, B., and Miranker, D. P. Effects of database size on rule system performance: five case studies. Proceedings of the International conference on very large databases, 1991.

6. Brownston, L., Farrell, R., Kant, E. and Martin, N. *Programming expert systems in OPS5: An introduction to rule-based programming.* Addison-Wesley, Reading, Massachusetts, 1985.

7. Buneman, P. and Clemons, E. "Efficiently monitoring relational databases". *ACM Transactions on Database Systems* (September 1979).

8. Delcambre, L. and Etheredge, J. N. The Relational Production Language: A Production Language for Relational Databases. Proceedings of the Second International Conference on Expert Database Systems, 1988, pp. 333-352.

9. Easwaran, K. Specification, implementation and interactions of a rule subsystem in an integrated database system. IBM Research Report, RJ1820, August, 1976.

10. Forgy, C. L. "Rete: A fast algorithm for the many pattern/many object pattern match problem". *Artificial Intelligence 19*, 1 (1982), 17-37.

11. Forgy, C. L. The OPS83 Report. Tech. Rept. CMU-CS-84-133, Computer Science Department, Carnegie Mellon University, 1984.

12. Gordin, D. N. and Pasik, A. J. Set-Oriented constructs: from Rete rule bases to database systems. Proceedings of the ACM SIGMOD conference on management of data, 1991, pp. 60-67.

13. Gupta, A. *Parallelism in production systems.* Ph.D. Th., Computer Science Department, Carnegie Mellon University, 1986. Also a book, Morgan Kaufmann, (1987)..

14. Gupta, A., Forgy, C., Newell, A., and Wedig, R. Parallel algorithms and architectures for production systems. Proceedings of the Thirteenth International Symposium on Computer Architecture, June, 1986, pp. 28-35.

15. Gupta, A., Tambe, M., Kalp, D., Forgy, C. L., and Newell, A. "Parallel implementation of OPS5 on the Encore Multiprocessor: Results and analysis". *International Journal of Parallel Programming 17*, 2 (1988).

16. Hanson, E. N. Rule condition testing and action execution in Ariel. Proceedings of the ACM SIGMOD conference on management of data, 1992, pp. 49-58.

17. *INGRES Version 6.3 Reference Manual.* 1990. INGRES Products Division, Alameda, CA.

18. Kalp, D. Tambe, M., Gupta, A., Forgy, C., Newell, A., Acharya, A., Milnes, B., and Swedlow, K. Parallel OPS5 User's Manual. Tech. Rept. CMU-CS-88-187, Computer Science Department, Carnegie Mellon University, November, 1988.

19. Laird, J. E., Newell, A. and Rosenbloom, P. S. "Soar: An architecture for general intelligence". *Artificial Intelligence 33*, 1 (1987), 1-64.

20. McKeown, D.M., Harvey, W.A., and McDermott, J. "Rule based interpretation of aerial imagery". *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 5 (1985), 570-585.

21. Miranker, D. P. Treat: A better match algorithm for AI production systems. Proceedings of the Sixth National Conference on Artificial Intelligence, 1987, pp. 42-47.

22. Miranker, D., and Lofaso, B. "The organization and performance of a Treat-based production system compiler". *IEEE transactions on knowledge and data engineering 3*, 1 (1991), 3-11.

23. Miranker, D. P., Brant, D. A., Lofaso, B., Gadbois, D., On the performance of lazy matching in production systems. Proceedings of the eigth national conference on artificial intelligence, 1990, pp. 685-692.

24. Nayak, P., Gupta, A. and Rosenbloom, P. Comparison of the Rete and Treat production matchers for Soar (A summary). Proceedings of the Seventh National Conference on Artificial Intelligence, 1988, pp. 693-698.

25. Scales, D.J. Efficient matching algorithms for the SOAR/OPS5 production system. Tech. Rept. KSL-86-47, Knowledge Systems Laboratory, Stanford University, June, 1986.

26. Sellis, T., and Lin, C. Performance of DBMS implementations of production systems. Proceedings of the International conference on tools for AI, 1990.

27. Sellis, T., Lin, C., and Raschid, L. "Data intensive production systems: The DIPS approach". *SIGMOD Record 18*, 3 (September 1989), 52-58.

28. Stonebraker, M. "The integration of rule systems with database systems". *IEEE Transactions on Knowledge and Data Engineering 4*, 5 (October 1992), 415-423.

29. *Sybase V4.0 Reference Manual.* 1990. Sybase Corp. Emeryville CA.

30. Tambe, M. *Eliminating combinatorics from production match.* Ph.D. Th., School of Computer Science, Carnegie Mellon University, May 1991.

31. Tambe, M. and Rosenbloom, P. A framework for investigating production system formulations with polynomially bounded match. Proceedings of the Eighth National Conference on Artificial Intelligence, 1990, pp. 693-400.

32. Tambe, M., Kalp, D., and Rosenbloom, P. An Efficient Match Algorithm for Unique-attribute Production Systems. Proceedings of the International Conference on Tools of Artificial Intelligence, 1992. (to appear).

33. Tan, J. S., Maheshwari, M., and Srivastava, J. GridMatch: A basis for integrating production systems with relational databases. Proceedings of the IEEE conference on Tools for AI, 1990, pp. 400-407.

34. Widom, J. and Finkelstein, S. "A Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems". *SIGMOD Record 18*, 3 (September 1989), 36-45.